1. Tempo disponibile 120 minuti.
2. Non è possibile consultare appunti, slide, libri, persone, siti web, ecc.
3. Scrivere in modo leggibile, su ogni foglio, nome, cognome e numero di matricola.
4. Le soluzioni agli esercizi che richiedono di progettare un algoritmo devono:
   - spiegare a parole l'algoritmo (se utile, anche con l'aiuto di esempi o disegni),
   - fornire e commentare lo pseudo-codice (indicando il significato delle variabili),
   - calcolare la complessità (con tutti i passaggi matematici necessari),
   - se l'esercizio ammette più soluzioni, a soluzioni computazionalmente più efficienti e/o concettualmente più semplici sono assegnati punteggi maggiori.

1. Calcolare la complessità $T(n)$ del seguente algoritmo `mystery1`:

```
algoritmo mystery1(n: Int) --> Int
  if (n <= 1)
    return 123
  else
    k = mystery2(n/2) + mystery1(n/3)
    return k + mystery1(n/3)
  endif

algoritmo mystery2(m: Int) --> Int
  if (m == 0)
    return 321
  else
    return 2 * mystery2(m/4) - mystery2(m/4)
  endif
```

Soluzione L'algoritmo `mystery1` invoca `mystery2`. Iniziamo quindi a calcolare il tempo di calcolo $T'(m)$ di `mystery2`. La funzione `mystery2` è ricorsiva con costo $T'(m)$ che soddisfa la seguente equazione di ricorrenza:

$$T'(m) = \begin{cases} 1 & \text{se } n \leq 1 \\ 2T'(m/4) + 1 & \text{altrimenti} \end{cases}$$

Applicando il master theorem otteniamo: $\alpha = \frac{\log a}{\log b} = \frac{\log 2}{\log 4} = \frac{1}{2}$ e $\beta = 0$. Avendo $\alpha > \beta$, applichiamo il primo caso del master theorem ottenendo $T'(m) = O(m^\alpha) = O(m^{\frac{1}{2}})$. Ora consideriamo il costo $T(n)$ della funzione ricorsiva `mystery1` che soddisfa la seguente equazione di ricorrenza:

$$T(n) = \begin{cases} 1 & \text{se } n \leq 1 \\ 2T(n/3) + (n/2)^{\frac{1}{2}} & \text{altrimenti} \end{cases}$$

Ma $(n/2)^{\frac{1}{2}}$ coincide con $(1/2)^{\frac{1}{2}} \times n^{\frac{1}{2}} = cn^{\frac{1}{2}}$, con c costante. Possiamo quindi applicare il master theorem ottenendo: $\alpha = \frac{\log a}{\log b} = \frac{\log 2}{\log 3} = \frac{1}{\log 3}$ e $\beta = \frac{1}{2}$. Avendo $\alpha > \beta$, applichiamo il primo caso del master theorem ottenendo $T(n) = O(n^\alpha) = O(n^{\frac{1}{\log 3}})$.

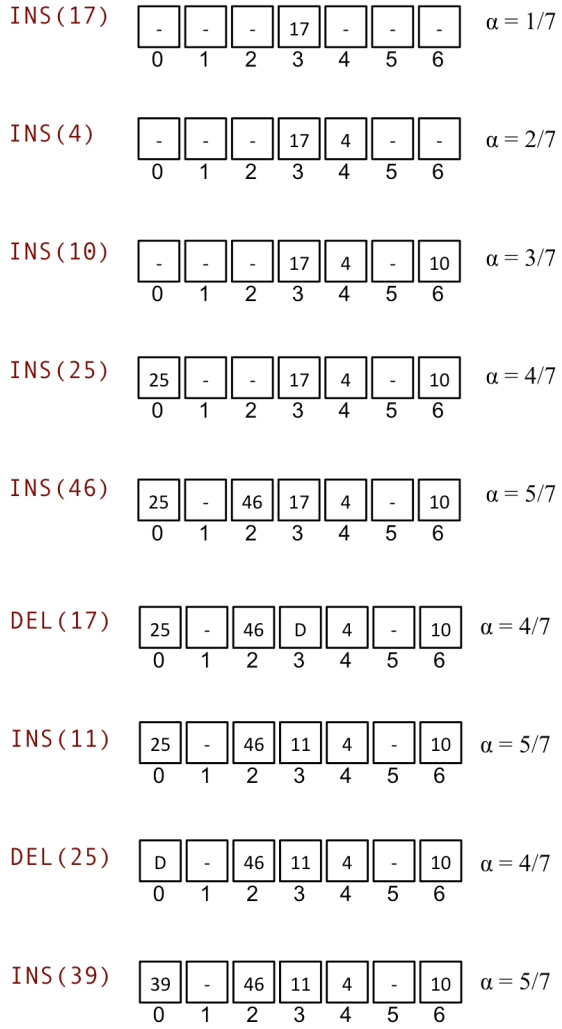
2. Si consideri una tabella hash di dimensione $m = 7$ inizialmente vuota. Le collisioni sono gestite mediante indirizzamento aperto usando la seguente funzione hash $h(k)$:

   $h(k) = (h'(k) + 3i) \mod m$

$h'(k) = k \mod m$

Si mostri il contenuto della tabella e il fattore di carico dopo ognuna delle seguenti operazioni, eseguite in ordine: INS(17), INS(4), INS(10), INS(25), INS(46), DEL(17), INS(11), DEL(25), INS(39).

Soluzione La seguente immagine mostra la tabella hash e il fattore di carico (n/m) dopo ogni operazione:



3. Si consideri un vettore bidimensionale $T[1..n, 1..n]$ di numeri, tale che $T[i, j] \leq T[i, j+1]$ e $T[i, n] \leq T[i+1, 1]$ (ovvero, ogni riga è ordinata in modo non decrescente, ed ogni riga contiene numeri minori o uguali a quelli della riga successiva). Progettare un algoritmo che dato il vettore bidimensionale T ed un numero n verifica se tale numero n è presente in T (ovvero restituisce un booleano, *true* se n è presente in T, *false* altrimenti).

Soluzione La matrice potrebbe essere vista logicamente come un vettore monodimensionale di lunghezza n^2 contente i valori della matrice letti riga per riga. In questo modo, tale vettore risulta ordinato in modo non decrescente, e quindi la ricerca di un valore può essere effettuata tramite ricerca binaria. Tale algoritmo è scritto successivamente, e viene inizialmente invocato con `RicBinaria(T, n, 1, n × n)`. La funzione `RicBinaria` prevede 4 parametri: la matrice T ed il valore da cercare n, e gli indici iniziale s e finale e indicanti la porzione di valore in cui ricercare il valore n.

```
algoritmo RicBinaria(T: Int[1..n,1..n], n: Int, s:Int, e:Int) --> Bool
  if (s>e) then return false
```

Cont.

```
else
  Int m = floor( (s+e)/2 )
  if (T[m/n + 1, m % n] == n) then return true
  else if (T[m/n + 1, m % n] < n) then
    return RicBinaria(T, n, m+1, e)
  else
    return RicBinaria(T, n, s, m-1)
```

Il tempo di calcolo $T(n)$, se si assume di esprimerlo in funzione del parametro n, sarà $T(n) = O(\log n^2)$ in quanto si tratta di una ricerca binaria su n^2 valori. Ma abbiamo, più semplicemente, $T(n) = O(\log n)$ in quanto $O(\log n^2) = O(2 \times \log n) = O(\log n)$.

4. Si consideri il seguente gioco, giocato con un mazzo di 40 carte tutte diverse fra loro, che per comodità identificheremo con i numeri nell'intervallo chiuso $[1, \ldots, 40]$. Inizialmente si prende una carta e si mettono le restanti 39 sul tavolo con la propria faccia in alto. Il gioco consiste nel formare coppie, abbinando una carta già presa con una carta ancora sul tavolo. Una volta abbinata, la carta sul tavolo viene presa dal tavolo. Si noti che le carte prese dal tavolo possono essere utilizzate più volte per formare abbinamenti con carte ancora sul tavolo. Dopo 39 abbinamenti, non ci saranno più carte sul tavolo ed il gioco termina. L'abbinamento di due carte $c_1, c_2 \in [1, \ldots, 40]$ contribuisce con un punteggio $P(\{c_1, c_2\})$. Il punteggio finale coincide con la somma dei punteggi dei 39 abbinamenti fatti. Tali punteggi possono essere rappresentati tramite una matrice bidimensionale P, tale che $P[c_1, c_2] = P[c_2, c_1]$. Progettare un algoritmo che data la matrice P, e la carta iniziale $s \in [1, \ldots, 40]$, restituisce il punteggio finale massimo che può essere ottenuto alla fine del gioco.

Soluzione La matrice bidimensionale P può essere vista come matrice dei costi degli archi di un grafo completo con vertici identificati dai numeri nell'intervallo chiuso $[1, \ldots, 40]$ (ovvero, un grafo in cui ogni vertice x è adiacente a tutti gli altri vertici in $[1, \ldots, 40] \setminus \{x\}$). Il gioco consiste nel definire delle coppie tra le carte, e quindi scegliere degli archi in tale grafo. Gli archi scelti definiscono un albero di copertura in quanto: (i) tutti i vertici sono coinvolti e (ii) non possono essere selezionati cicli (in quanto un ciclo coinciderebbe con il prelevare due volte dal tavolo la medesima carta). Quindi il problema può essere risolto andando alla ricerca di un maximum spanning tree sul grafo rappresentato dalla matrice P. Presentiamo una soluzione che adotta una versione adattata dell'algoritmo di Prim, in cui si usa una coda con priorità massima invece che minima, e che ritorna il costo complessivo del maximum spanning tree calcolato.

```
algoritmo MaximumSpanningTree (P: Int[1..40,1..40], s: Int) --> Double

  // inizializzazione strutture dati
  Double tot = 0
  Double D[1..40]
  for v = 1..40 do D[v] = -INFINITY
  D[s] = 0
  CodaPriorita<Int, Double> Q
  Q.insert(s, D[s])

  // esecuzione algoritmo di Prim
  while (not Q.isEmpty()) do
    u = Q.find(); Q.deleteMax()
    tot = tot + D[u]
    for each v adiacente a u do
      if (D[v] == -INFINITY) then
        // prima volta che si incontra v
        D[v] = P[u,v]
        Q.insert(v, D[v]);
      elseif (P[u,v] > D[v]) then
        // scoperta di un arco migliore per raggiungere v
        Q.increaseKey(v, P[u,v] - D[v])
        D[v] = P[u,v]
      endif
```

Cont.

```
  endfor
endwhile

// restituisce il punteggio totale
return tot
```

Il costo computazionale coincide con quello dell'algoritmo di Prim, ovvero $O(m \log n)$, con m numero di archi ed n numero di vertici. Essendo il grafo completo, il numero di archi è $n \times (n-1) = O(n^2)$. Quindi il costo sarebbe $O(n^2 \log n)$. Ma, essendo la dimensione dell'input costante in quanto $n = 40$, possiamo concludere che in questo caso l'analisi del costo computazionale asintotico non ha senso in quanto l'input non è previsto che possa crescere.

The End.